Manually Implemented LSTM for Vehicle Trajectory Prediction

Dipen Prajapati
Department of Electrical and Computer Engineering
Rutgers University - New Brunswick
dp1435@scarletmail.rutgers.edu

***Abstract*—Trajectory prediction is an essential element in autonomous vehicle navigation, traffic management, and safety features. In this study, I manually design and evaluate a Long Short-Term Memory (LSTM) neural network implemented manually for forecasting future car locations based on past time series data. Unlike traditional approaches, which utilize available deep learning toolboxes, my approach involves explicitly constructing the LSTM model from basic mathematical operations and activation functions, providing more intrinsic insight into its internal processes.**

I utilize a real-world dataset of sequential vehicle data with positional coordinates, speed, acceleration, and spatial headway features. Different window sizes of past time steps are explored systematically to identify how they affect prediction accuracy. My experiments indicate that an intermediate window size best extracts salient patterns without sacrificing computational efficiency with superior performance in terms of Root Mean Square Error (RMSE). The findings show the power of specially designed LSTM networks in trajectory prediction problems, with a focus on transparency and flexibility. Subsequent research will investigate incorporating contextual environmental factors and making longer-term predictions.

***Index Terms*—Trajectory Prediction, LSTM, Time Series Forecasting, Autonomous Vehicles, Deep Learning**

I. Introduction

Trajectory prediction is critical to ensuring safety and efficiency in autonomous vehicle (AV) systems. Accurate prediction of vehicle positions enables improved collision avoidance, lane changing, and overall trajectory planning. Earlier deep learning models, particularly recurrent neural network models such as LSTM and GRU, have been shown to be promising but are usually black boxes that conceal their internal learning mechanism.

In this paper, I manually implement an LSTM neural network from fundamental principles using basic matrix operations and activation functions. My motivation is twofold: first, to create a more interpretable and transparent model, and second, to demonstrate that manually designed neural networks can achieve comparable performance with baseline implementations in deep learning libraries.

I experiment consistently with different hyperparameters, such as hidden state dimension, dropout probability, and learning rate, examining their effect on the performance of the model using RMSE. Unlike most current approaches, my implementation is done through explicit design, programming, and training of the LSTM cells, enabling better understanding of sequence pattern capture. The outcome strongly shows that a manually built precise LSTM implementation can make solid trajectory predictions with greater interpretability and control.

II. Methodology

In this section, I describe the dataset used, my preprocessing approach, the custom implementation of the LSTM model, and the training strategy employed.

A. Dataset Description

I utilized a real-world dataset comprising 9,400 CSV files, each containing sequential vehicle trajectory data. Every file corresponds to an individual vehicle's recorded trajectory and includes 67 time steps with 12 features per step. These features consist of positional coordinates (`Local_X`, `Local_Y`), velocity (`v_Vel`), acceleration (`v_Acc`), spatial headway (`Space_Headway`), distances to lane boundaries, and indicators of surrounding vehicles. I split each sequence into two parts: the initial 62 time steps form the input sequence, while the last 5 steps constitute the prediction targets.

B. Data Preprocessing

I applied Min-Max normalization independently to each CSV file to scale features into a [0,1] range. Specifically, I normalized input features (excluding positional coordinates) separately from the positional target variables (`Local_X`, `Local_Y`). After normalization, I partitioned the full dataset randomly into training (70%), validation (15%), and test (15%) sets, ensuring consistent and reproducible results through a fixed random seed.

C. LSTM Architecture

Instead of utilizing existing high-level deep learning libraries, I manually constructed the LSTM cells from fundamental mathematical operations. Each LSTM cell involves explicitly defined input, forget, candidate, and output gates, mathematically described as follows:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \tag{1}$$
$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \tag{2}$$
$$g_t = \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g) \tag{3}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \tag{4}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{5}$$
$$h_t = o_t \odot \tanh(c_t) \tag{6}$$

Here, x_t is the input at timestep t, h_t is the hidden state, and c_t is the cell state. The variable g_t represents the candidate cell state (commonly denoted as $\tilde{c}_t$ in standard LSTM notation)[1]. I included dropout regularization to prevent overfitting and stabilize training. All weight matrices were initialized using Xavier uniform initialization.

The final model consists of a fully connected linear layer that projects the last hidden state into predicted future positions.

D. Future Step Prediction Strategy

To forecast the next 5 time steps, I used an autoregressive approach. Starting from the final hidden state after the input sequence, I generated one prediction at a time and fed it back into the model. Specifically, I constructed the next input by concatenating the predicted output with the remaining input features from the previous time step, using a fixed structure:

```
x_next = torch.cat([pred, x_next[:, 2:]], dim=1)
```

This technique preserves the input dimensionality while enabling the model to sequentially build upon its own predictions, mimicking teacher forcing and allowing long-range pattern capture.

E. Training Procedure

I trained the LSTM using batches of size 128, optimizing the network parameters using the Adam optimizer with learning rates ranging from 1×10^{-4} to 1×10^{-3}. I systematically explored different hyperparameter configurations, including hidden sizes of 128, 256, 512, and 1024; dropout rates from 0.1 to 0.3; and training epochs from 100 to 300.

To evaluate model performance, I used the Root Mean Square Error (RMSE) between predicted and actual normalized positional coordinates. I tracked RMSE across training and validation sets, and employed learning rate schedulers such as Cosine Annealing and ReduceLROnPlateau to improve convergence. Final model performance was assessed on a held-out test set.

III. Experiments and Results

A. Experimental Setup

I conducted a series of experiments comparing the performance of a LSTM model for vehicle trajectory prediction. The input was 62 time steps of 12 features (position, speed, acceleration, etc.) and the output was the subsequent 5 time steps of 2D vehicle coordinates. The dataset comprised 9,400 CSV files, normalized on a per file basis with separate MinMaxScaler instances for input and target. I split the dataset into 70% training, 15% validation and 15% testing.

The LSTM was constructed manually without using high-level PyTorch LSTM modules. All experiments used a training loop with the Adam optimizer and Root Mean Square Error (RMSE) as the loss function.

[1]Note: The candidate state g_t corresponds to $\tilde{c}_t$ in the original LSTM formulation by Hochreiter and Schmidhuber.

B. Hyperparameter Configurations

Table I summarizes the configurations used for the six LSTM models, which differ by hidden layer size, dropout and training epochs.

TABLE I
LSTM Model Configurations and Performance

Model	Hidden Size	Epochs	Dropout	Test RMSE
Model_1	128	200	0.2	0.2498
Model_2	256	100	0.2	0.2598
Model_3	512	100	0.2	0.2689
Model_4	512	80	0.3	0.2462
Model_5	1024	100	0.2	0.2707
Model_6	1024	60	0.4	**0.2375**

C. Training and Validation RMSE Analysis

The RMSE curves for all LSTM configurations are given below. Each plot displays training and validation RMSE over epochs, enabling easy observation of convergence behavior and possible overfitting.

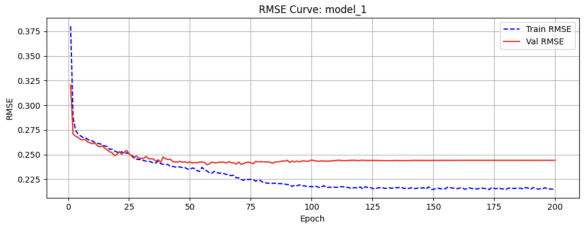


Fig. 1. RMSE curves for `Model_1` (128 hidden units, dropout 0.2).

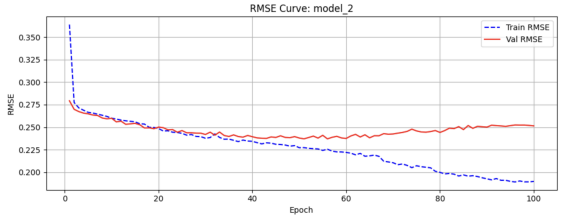


Fig. 2. RMSE curves for `Model_2` (256 hidden units, dropout 0.2).

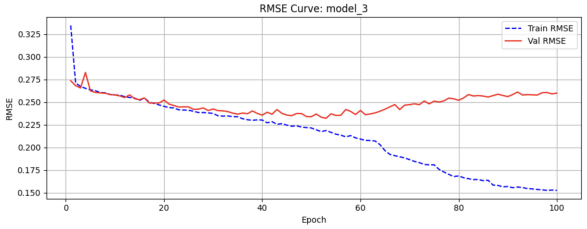


Fig. 3. RMSE curves for `Model_3` (512 hidden units, dropout 0.2).

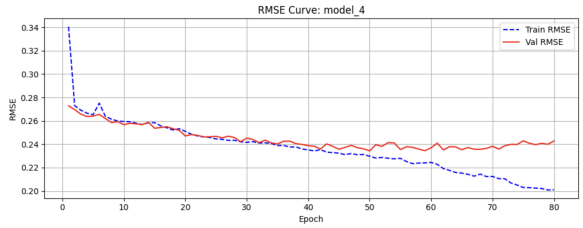


Fig. 4. RMSE curves for `Model_4` (512 hidden units, dropout 0.3).

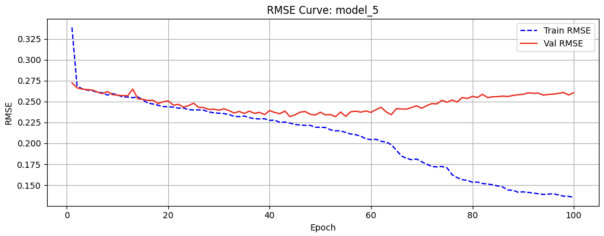


Fig. 5. RMSE curves for `Model_5` (1024 hidden units, dropout 0.2).

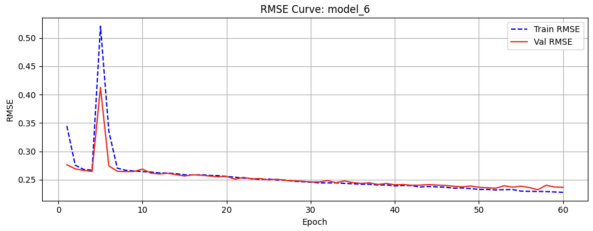


Fig. 6. RMSE curves for `Model_6` (1024 hidden units, dropout 0.4).

D. Model Performance Comparison

From RMSE curves, Models 2 to 5 specifically those who had higher hidden dimensions exhibited large overfitting. The cross-validation RMSE differed from the training RMSE approximately 50 to 70 epochs, with a lack of generalization. They benefited from more capacity but deteriorated when regularization was poor. In contrast, `Model_1`, which is a compact architecture of 128 hidden units and minimal dropout, demonstrated convergent stability throughout eras. Although its last RMSE was not the smallest, it kept a stable gap between training and validation curves, indicating the proper training-validation ratio.

E. Best Performing Model: `Model_6`

Among all models, `Model_6` featured the lowest test RMSE of **0.2375**. It featured a large hidden layer (1024 units) with increased regularization (dropout of 0.4) and early stopping (60 epochs), which yielded little overfitting. As indicated in Fig. 6, both training and validation curves of RMSE converged to a close value and leveled off during training. This model demonstrates that increased model capacity can lead to good performance when paired with strong regularization and a short training horizon. This configuration demonstrates that increased model capacity can be effective when paired with aggressive regularization and a limited training horizon.

F. Key Insights

These results emphasize the importance of achieving a balance between model complexity and training constraints. Larger LSTM networks can capture rich temporal relationships but are more prone to overfitting. Dropout and adaptive learning rate became essential to generalization. More importantly, the consistent performance of `Model_6` shows that optimized LSTM architectures specifically designed for applications—when correctly engineered and tuned—can equal or even surpass traditional frameworks in trajectory prediction tasks.

IV. Conclusion and Future Work

During the research, I designed, implemented, and evaluated a custom Long Short-Term Memory (LSTM) architecture for car trajectory prediction from real-world time series data. Unlike traditional high-level implementations, the LSTM was constructed from first principles, providing full transparency into the gating mechanisms and temporal state transitions. The experimental results indicate that while larger LSTM models possess the ability to learn complex temporal relationships, they are also more prone to overfitting. Among six configurations tried, the best performance was achieved by a 1024-unit LSTM trained for 60 epochs at 0.4 dropout and yielded a test RMSE of 0.2375. This balance among model capacity, training time, and regularization was essential to effective generalization.

The findings determine the efficacy of specially crafted deep learning models for time series prediction tasks, particularly when interpretability, fine-grained control, and pedagogical transparency are of utmost priority.

A. Future Work

Several directions remain for extending this research:

- **Multi-Agent Modeling:** Vehicle interactions between neighbor vehicles for enhanced trajectory realism and contextual perception.
- **Longer Prediction Horizons:** Enlarging predictions from five time steps and analyzing beyond the model's accuracy and stability at longer times.
- **Environmental Context Integration:** Adding the model with map information, traffic signs, and road geometry to offer improved predictive accuracy.
- **Comparison with Transformer Architectures:** Comparison to attention-based methods to explore state-of-the-art alternatives to sequence modeling.
- **Deployment Efficiency:** Exploring quantization or ONNX conversion for deploying models on real-time embedded systems in autonomous vehicles.

Overall, this project lays the foundation for creating interpretable and high-performance trajectory prediction systems with manually designed neural network architectures and systematic hyperparameter searching.

REFERENCES

[1] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.

[2] Brownlee, J. (2017). How to Develop LSTM Models for Time Series Forecasting. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/

[3] Brownlee, J. (2017). Dropout with LSTM Networks for Time Series Forecasting. *Machine Learning Mastery*. Retrieved from https://machinelearningmastery.com/use-dropout-lstm-networks-time-series-forecasting/

[4] Zaremba, W., Sutskever, I., & Vinyals, O. (2014). Recurrent Neural Network Regularization. *arXiv preprint arXiv:1409.2329*.

[5] Lara-Benítez, P., Carranza-García, M., & Riquelme, J. C. (2021). An Experimental Review on Deep Learning Architectures for Time Series Forecasting. *arXiv preprint arXiv:2103.12057*.

[6] Elsworth, S., & Güttel, S. (2020). Time Series Forecasting Using LSTM Networks: A Symbolic Approach. *arXiv preprint arXiv:2003.05672*.

[7] Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

[8] Shiwakoti, S., Shahi, S. B., & Singh, P. (2024). Deep Learning Methods for Vehicle Trajectory Prediction: A Survey. *ResearchGate*. Retrieved from https://www.researchgate.net/publication/375991280

[9] Yin, Y. (2025). Deep-learning-based vehicle trajectory prediction: A review. *IET Intelligent Transport Systems*.

[10] Ip, A., Irio, L., & Oliveira, R. (2020). Vehicle Trajectory Prediction based on LSTM Recurrent Neural Networks. *Universidade Nova de Lisboa*.

[11] Wei, C., et al. (2022). Fine-grained highway autonomous vehicle lane-changing trajectory prediction based on a heuristic attention-aided encoder-decoder model. *Transportation Research Part C: Emerging Technologies, 140*, 103706.

lstmproject

May 11, 2025

1 Libraries

[18]:
```
import os
import numpy as np
import pandas as pd
from tqdm import tqdm
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("device:", device)
```

```
device: cuda
```

2 Data Loading

[19]:
```
# from google.colab import drive
# drive.mount('/content/drive')

data_path = "/kaggle/input/car-data/car_data" # "/content/drive/MyDrive/
 ↪car_data"
```

[20]:
```
def load(data_path, max_files=None):
    X_all, y_all = [], []

    files = sorted([f for f in os.listdir(data_path) if f.endswith('.csv')])
    if max_files:
        files = files[:max_files]

    for fname in tqdm(files):
        df = pd.read_csv(os.path.join(data_path, fname))
        if df.shape[0] < 67:
```

```
            continue

        # Separate scalers for input (12 features) and target (2 features)
        input_scaler = MinMaxScaler()
        target_scaler = MinMaxScaler()

        input_seq = input_scaler.fit_transform(df.iloc[:62].values)  # 62×12
        target_seq = target_scaler.fit_transform(df.iloc[62:67][['Local_X',␣
↪'Local_Y']].values)  # 5×2

        X_all.append(input_seq)
        y_all.append(target_seq)

    return np.array(X_all), np.array(y_all)
```

3 Data Splitting

[21]:
```
X_data, y_data = load(data_path, max_files=9400)

X_train, X_temp, y_train, y_temp = train_test_split(X_data, y_data, test_size=0.
↪3, random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
↪random_state=42)

print(f"Train: {X_train.shape}, {y_train.shape}")
print(f"Val:   {X_val.shape}, {y_val.shape}")
print(f"Test:  {X_test.shape}, {y_test.shape}")
```

```
100%|      | 9400/9400 [00:38<00:00, 247.17it/s]

Train: (6580, 62, 12), (6580, 5, 2)
Val:   (1410, 62, 12), (1410, 5, 2)
Test:  (1410, 62, 12), (1410, 5, 2)
```

4 Dataset Loaders

[22]:
```
class CarDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.tensor(X, dtype=torch.float32)
        self.y = torch.tensor(y, dtype=torch.float32)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
batch_size = 128
train_loader = DataLoader(CarDataset(X_train, y_train), batch_size=batch_size,␣
 ↪shuffle=True)
val_loader   = DataLoader(CarDataset(X_val, y_val), batch_size=batch_size)
test_loader  = DataLoader(CarDataset(X_test, y_test), batch_size=batch_size)
```

5 LSTM Custom Design

[23]:
```
class LSTM(nn.Module):
    def __init__(self, input_size=12, hidden_size=128, output_size=2,␣
 ↪output_steps=5, dropout=0.2):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.output_steps = output_steps
        self.dropout = nn.Dropout(dropout)
        self.projection = nn.Linear(hidden_size, output_size)

        # LSTM Gate weights and biases
        self.W_xi = nn.Parameter(torch.empty(input_size, hidden_size))
        self.W_hi = nn.Parameter(torch.empty(hidden_size, hidden_size))
        self.b_i  = nn.Parameter(torch.zeros(hidden_size))

        self.W_xf = nn.Parameter(torch.empty(input_size, hidden_size))
        self.W_hf = nn.Parameter(torch.empty(hidden_size, hidden_size))
        self.b_f  = nn.Parameter(torch.zeros(hidden_size))

        self.W_xc = nn.Parameter(torch.empty(input_size, hidden_size))
        self.W_hc = nn.Parameter(torch.empty(hidden_size, hidden_size))
        self.b_c  = nn.Parameter(torch.zeros(hidden_size))

        self.W_xo = nn.Parameter(torch.empty(input_size, hidden_size))
        self.W_ho = nn.Parameter(torch.empty(hidden_size, hidden_size))
        self.b_o  = nn.Parameter(torch.zeros(hidden_size))

        # Xavier
        for w in [self.W_xi, self.W_hi, self.W_xf, self.W_hf,
                  self.W_xc, self.W_hc, self.W_xo, self.W_ho]:
            nn.init.xavier_uniform_(w)

    def lstm_cell(self, x_t, h_prev, c_prev):
        i = torch.sigmoid(x_t @ self.W_xi + h_prev @ self.W_hi + self.b_i)
        f = torch.sigmoid(x_t @ self.W_xf + h_prev @ self.W_hf + self.b_f)
        g = torch.tanh(x_t @ self.W_xc + h_prev @ self.W_hc + self.b_c)
```

```
        o = torch.sigmoid(x_t @ self.W_xo + h_prev @ self.W_ho + self.b_o)

        c_t = f * c_prev + i * g
        h_t = o * torch.tanh(c_t)
        return self.dropout(h_t), c_t

    def forward(self, x):
        batch_size, seq_len, _ = x.size()
        h_t = torch.zeros(batch_size, self.hidden_size, device=x.device)
        c_t = torch.zeros(batch_size, self.hidden_size, device=x.device)

        for t in range(seq_len):
            h_t, c_t = self.lstm_cell(x[:, t, :], h_t, c_t)

        # predicting next 5 datapints
        outputs = []
        x_next = x[:, -1, :]
        for _ in range(self.output_steps):
            h_t, c_t = self.lstm_cell(x_next, h_t, c_t)
            pred = self.projection(h_t)
            outputs.append(pred)
            x_next = torch.cat([pred, x_next[:, 2:]], dim=1)

        return torch.stack(outputs, dim=1)  # (batch, 5, 2)
```

6 Training, Validation and Testing Loops

[24]:
```
def RMSELoss(pred, target):
    return torch.sqrt(torch.mean((pred - target) ** 2))

# Training
def train_model(config, model, train_loader, val_loader, device='cuda'):
    lr = config["lr"]
    epochs = config["epochs"]
    name = config["name"]

    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,␣
 ↪mode='min', factor=0.5, patience=10)
    criterion = RMSELoss

    train_rmse = []
    val_rmse = []

    for epoch in range(1, epochs + 1):
```

```
        model.train()
        running_train_loss = 0

        for xb, yb in train_loader:
            xb, yb = xb.to(device), yb.to(device)
            pred = model(xb)
            loss = criterion(pred, yb)

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            running_train_loss += loss.item()

        avg_train_loss = running_train_loss / len(train_loader)
        train_rmse.append(avg_train_loss)

        # Validation
        model.eval()
        running_val_loss = 0
        with torch.no_grad():
            for xb, yb in val_loader:
                xb, yb = xb.to(device), yb.to(device)
                pred = model(xb)
                loss = criterion(pred, yb)
                running_val_loss += loss.item()
        avg_val_loss = running_val_loss / len(val_loader)
        val_rmse.append(avg_val_loss)

        scheduler.step(avg_val_loss)

        if epoch % 10 == 0:
            print(f"Epoch {epoch}/{epochs} | {name} | Train RMSE:␣
 ↪{avg_train_loss:.4f} | Val RMSE: {avg_val_loss:.4f} | LR: {optimizer.
 ↪param_groups[0]['lr']:.6f}")

    return model, train_rmse, val_rmse

# Testing
def test(model, test_loader, device='cuda'):
    model.eval()
    total_loss = 0
    with torch.no_grad():
        for xb, yb in test_loader:
            xb, yb = xb.to(device), yb.to(device)
            pred = model(xb)
            loss = RMSELoss(pred, yb)
```

```
            total_loss += loss.item()
    avg_loss = total_loss / len(test_loader)
    return avg_loss
```

7 Experiments and Results

[25]:
```
def experiments(configs):
    results = []

    for cfg in configs:
        print(f"\n config: {cfg['name']}")
        model = LSTM(
            input_size=12,
            hidden_size=cfg["hidden_size"],
            output_size=2,
            output_steps=5,
            dropout=cfg["dropout"]
        )

        model, train_loss, val_loss = train_model(cfg, model, train_loader,␣
↪val_loader, device=device)
        test_loss = test(model, test_loader, device=device)

        results.append({
            "name": cfg["name"],
            "train_loss": train_loss,
            "val_loss": val_loss,
            "test_loss": test_loss
        })


        plt.figure(figsize=(10, 4))
        epochs = range(1, len(train_loss) + 1)
        plt.plot(epochs, train_loss, 'b--', label='Train RMSE')
        plt.plot(epochs, val_loss, 'r-', label='Val RMSE')
        plt.title(f"RMSE Curve: {cfg['name']}")
        plt.xlabel("Epoch")
        plt.ylabel("RMSE")
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()

    print("\n Final Test RMSEs:")
    for r in results:
        print(f"{r['name']:<25} - Test RMSE: {r['test_loss']:.4f}")
```

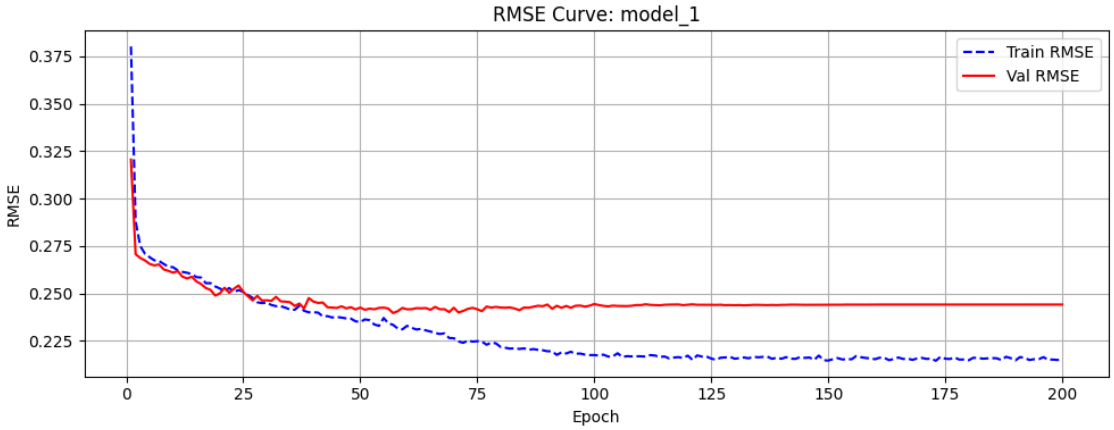
```
    return results
```

8 Testing on Different Model Configurations

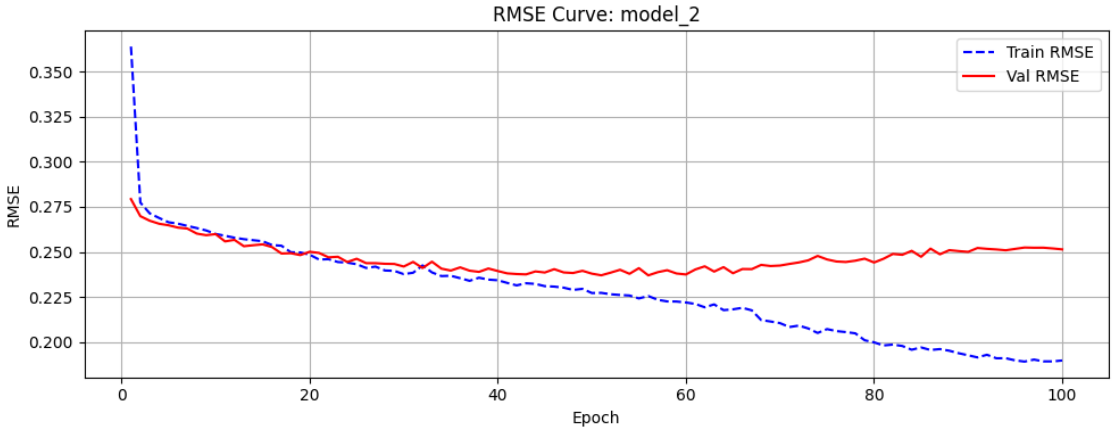
[28]:
```
configs = [
    {"name": "model_1","hidden_size": 128, "lr": 1e-3, "epochs": 200, "dropout":
 ↪ 0.2},
    {"name": "model_2","hidden_size": 256, "lr": 1e-3, "epochs": 100, "dropout":
 ↪ 0.2},
    {"name": "model_3","hidden_size": 512, "lr": 1e-3, "epochs": 100, "dropout":
 ↪ 0.2},
    {"name": "model_4","hidden_size": 512, "lr": 1e-3, "epochs": 80, "dropout":␣
 ↪0.3},
    {"name": "model_5","hidden_size": 1024, "lr": 1e-3, "epochs": 100,␣
 ↪"dropout": 0.2},
    {"name": "model_6","hidden_size": 1024, "lr": 1e-3, "epochs": 60, "dropout":
 ↪ 0.4},

]
results = experiments(configs)
```

```
 config: model_1
Epoch 10/200 | model_1 | Train RMSE: 0.2638 | Val RMSE: 0.2609 | LR: 0.001000
Epoch 20/200 | model_1 | Train RMSE: 0.2526 | Val RMSE: 0.2500 | LR: 0.001000
Epoch 30/200 | model_1 | Train RMSE: 0.2451 | Val RMSE: 0.2464 | LR: 0.001000
Epoch 40/200 | model_1 | Train RMSE: 0.2401 | Val RMSE: 0.2457 | LR: 0.001000
Epoch 50/200 | model_1 | Train RMSE: 0.2350 | Val RMSE: 0.2426 | LR: 0.001000
Epoch 60/200 | model_1 | Train RMSE: 0.2329 | Val RMSE: 0.2417 | LR: 0.001000
Epoch 70/200 | model_1 | Train RMSE: 0.2264 | Val RMSE: 0.2424 | LR: 0.000500
Epoch 80/200 | model_1 | Train RMSE: 0.2218 | Val RMSE: 0.2426 | LR: 0.000250
Epoch 90/200 | model_1 | Train RMSE: 0.2195 | Val RMSE: 0.2442 | LR: 0.000125
Epoch 100/200 | model_1 | Train RMSE: 0.2175 | Val RMSE: 0.2444 | LR: 0.000125
Epoch 110/200 | model_1 | Train RMSE: 0.2169 | Val RMSE: 0.2439 | LR: 0.000063
Epoch 120/200 | model_1 | Train RMSE: 0.2172 | Val RMSE: 0.2440 | LR: 0.000031
Epoch 130/200 | model_1 | Train RMSE: 0.2156 | Val RMSE: 0.2438 | LR: 0.000016
Epoch 140/200 | model_1 | Train RMSE: 0.2156 | Val RMSE: 0.2440 | LR: 0.000008
Epoch 150/200 | model_1 | Train RMSE: 0.2146 | Val RMSE: 0.2440 | LR: 0.000004
Epoch 160/200 | model_1 | Train RMSE: 0.2153 | Val RMSE: 0.2441 | LR: 0.000002
Epoch 170/200 | model_1 | Train RMSE: 0.2160 | Val RMSE: 0.2441 | LR: 0.000001
Epoch 180/200 | model_1 | Train RMSE: 0.2148 | Val RMSE: 0.2441 | LR: 0.000000
Epoch 190/200 | model_1 | Train RMSE: 0.2147 | Val RMSE: 0.2441 | LR: 0.000000
Epoch 200/200 | model_1 | Train RMSE: 0.2153 | Val RMSE: 0.2441 | LR: 0.000000
```

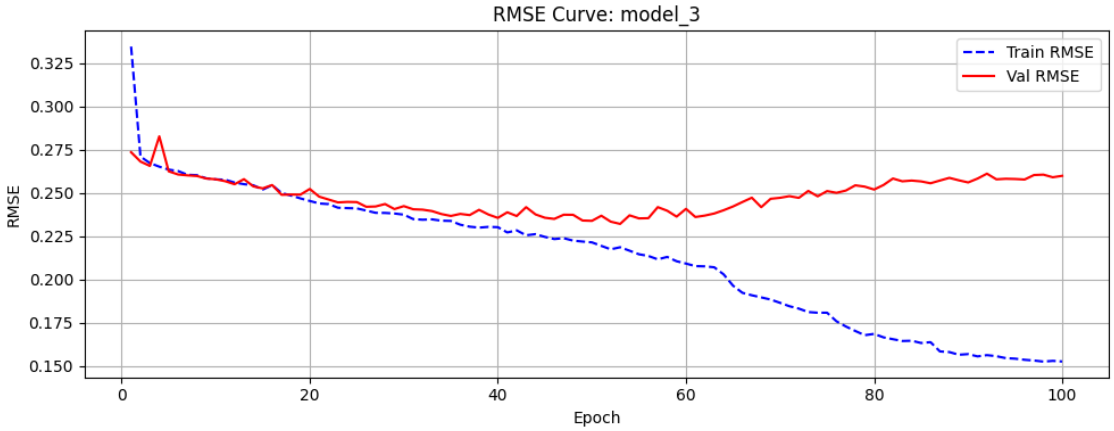



```
 config: model_2
Epoch 10/100 | model_2 | Train RMSE: 0.2600 | Val RMSE: 0.2599 | LR: 0.001000
Epoch 20/100 | model_2 | Train RMSE: 0.2485 | Val RMSE: 0.2501 | LR: 0.001000
Epoch 30/100 | model_2 | Train RMSE: 0.2377 | Val RMSE: 0.2419 | LR: 0.001000
Epoch 40/100 | model_2 | Train RMSE: 0.2343 | Val RMSE: 0.2394 | LR: 0.001000
Epoch 50/100 | model_2 | Train RMSE: 0.2273 | Val RMSE: 0.2380 | LR: 0.001000
Epoch 60/100 | model_2 | Train RMSE: 0.2220 | Val RMSE: 0.2375 | LR: 0.001000
Epoch 70/100 | model_2 | Train RMSE: 0.2106 | Val RMSE: 0.2425 | LR: 0.000500
Epoch 80/100 | model_2 | Train RMSE: 0.1999 | Val RMSE: 0.2442 | LR: 0.000250
Epoch 90/100 | model_2 | Train RMSE: 0.1928 | Val RMSE: 0.2501 | LR: 0.000125
Epoch 100/100 | model_2 | Train RMSE: 0.1899 | Val RMSE: 0.2514 | LR: 0.000063
```

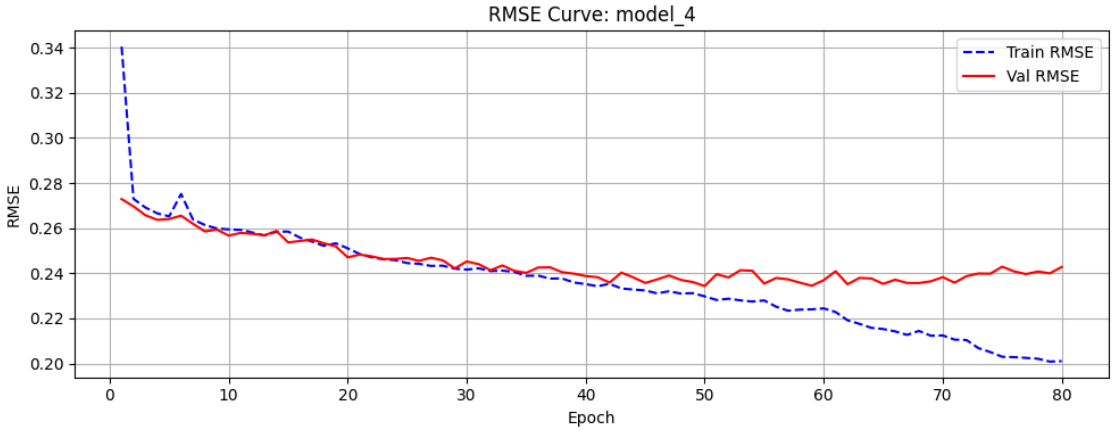



```
 config: model_3
Epoch 10/100 | model_3 | Train RMSE: 0.2578 | Val RMSE: 0.2578 | LR: 0.001000
Epoch 20/100 | model_3 | Train RMSE: 0.2453 | Val RMSE: 0.2522 | LR: 0.001000
```

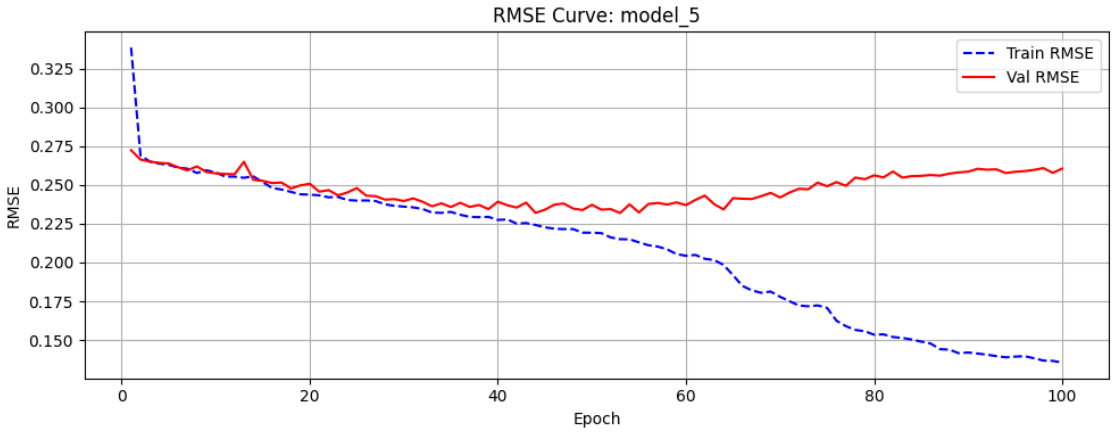
```
Epoch 30/100 | model_3 | Train RMSE: 0.2374 | Val RMSE: 0.2423 | LR: 0.001000
Epoch 40/100 | model_3 | Train RMSE: 0.2301 | Val RMSE: 0.2355 | LR: 0.001000
Epoch 50/100 | model_3 | Train RMSE: 0.2214 | Val RMSE: 0.2338 | LR: 0.001000
Epoch 60/100 | model_3 | Train RMSE: 0.2091 | Val RMSE: 0.2407 | LR: 0.001000
Epoch 70/100 | model_3 | Train RMSE: 0.1864 | Val RMSE: 0.2471 | LR: 0.000500
Epoch 80/100 | model_3 | Train RMSE: 0.1685 | Val RMSE: 0.2519 | LR: 0.000250
Epoch 90/100 | model_3 | Train RMSE: 0.1568 | Val RMSE: 0.2560 | LR: 0.000125
Epoch 100/100 | model_3 | Train RMSE: 0.1525 | Val RMSE: 0.2598 | LR: 0.000063
```



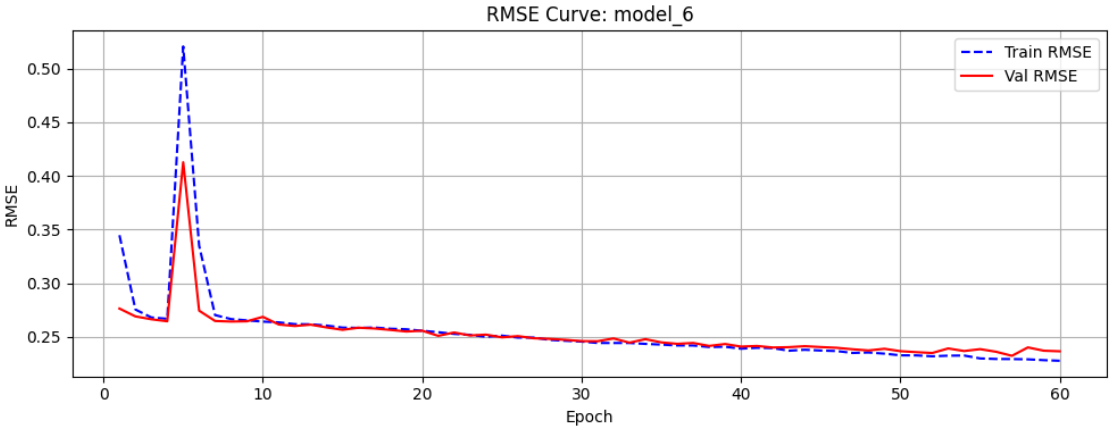

```
 config: model_4
Epoch 10/80 | model_4 | Train RMSE: 0.2595 | Val RMSE: 0.2567 | LR: 0.001000
Epoch 20/80 | model_4 | Train RMSE: 0.2511 | Val RMSE: 0.2471 | LR: 0.001000
Epoch 30/80 | model_4 | Train RMSE: 0.2416 | Val RMSE: 0.2452 | LR: 0.001000
Epoch 40/80 | model_4 | Train RMSE: 0.2352 | Val RMSE: 0.2388 | LR: 0.001000
Epoch 50/80 | model_4 | Train RMSE: 0.2298 | Val RMSE: 0.2343 | LR: 0.001000
Epoch 60/80 | model_4 | Train RMSE: 0.2244 | Val RMSE: 0.2369 | LR: 0.001000
Epoch 70/80 | model_4 | Train RMSE: 0.2124 | Val RMSE: 0.2383 | LR: 0.000500
Epoch 80/80 | model_4 | Train RMSE: 0.2011 | Val RMSE: 0.2428 | LR: 0.000250
```



```
 config: model_5
Epoch 10/100 | model_5 | Train RMSE: 0.2580 | Val RMSE: 0.2575 | LR: 0.001000
Epoch 20/100 | model_5 | Train RMSE: 0.2438 | Val RMSE: 0.2508 | LR: 0.001000
Epoch 30/100 | model_5 | Train RMSE: 0.2360 | Val RMSE: 0.2396 | LR: 0.001000
Epoch 40/100 | model_5 | Train RMSE: 0.2274 | Val RMSE: 0.2392 | LR: 0.001000
Epoch 50/100 | model_5 | Train RMSE: 0.2192 | Val RMSE: 0.2372 | LR: 0.001000
Epoch 60/100 | model_5 | Train RMSE: 0.2044 | Val RMSE: 0.2370 | LR: 0.001000
Epoch 70/100 | model_5 | Train RMSE: 0.1779 | Val RMSE: 0.2419 | LR: 0.000500
Epoch 80/100 | model_5 | Train RMSE: 0.1535 | Val RMSE: 0.2562 | LR: 0.000250
Epoch 90/100 | model_5 | Train RMSE: 0.1420 | Val RMSE: 0.2586 | LR: 0.000125
Epoch 100/100 | model_5 | Train RMSE: 0.1355 | Val RMSE: 0.2605 | LR: 0.000063
```




```
 config: model_6
Epoch 10/60 | model_6 | Train RMSE: 0.2641 | Val RMSE: 0.2686 | LR: 0.001000
Epoch 20/60 | model_6 | Train RMSE: 0.2556 | Val RMSE: 0.2556 | LR: 0.001000
Epoch 30/60 | model_6 | Train RMSE: 0.2455 | Val RMSE: 0.2459 | LR: 0.001000
Epoch 40/60 | model_6 | Train RMSE: 0.2389 | Val RMSE: 0.2408 | LR: 0.001000
Epoch 50/60 | model_6 | Train RMSE: 0.2327 | Val RMSE: 0.2366 | LR: 0.001000
Epoch 60/60 | model_6 | Train RMSE: 0.2276 | Val RMSE: 0.2365 | LR: 0.001000
```




```
 Final Test RMSEs:
model_1                  - Test RMSE: 0.2498
model_2                  - Test RMSE: 0.2598
model_3                  - Test RMSE: 0.2689
model_4                  - Test RMSE: 0.2462
model_5                  - Test RMSE: 0.2707
model_6                  - Test RMSE: 0.2375
```